# An Implementation of the Modular Integer GCD Algorithm on a Single GPU

KENNETH WEBER, University of Mount Union, USA
JUSTIN A. BREW, University of Mount Union, USA

The modular integer greatest common divisor (GCD) algorithm [12] holds promise to provide superior performance to sequential algorithms on extremely large input. In order to demonstrate the efficacy of the algorithm, an implementation on a system with multiple Graphics Processing Units (GPUs) is proposed, based on a single-GPU implementation described herein. The implementation's performance is analyzed to predict the size of input needed to demonstrate superior performance when compared to one popular sequential implementation of the integer GCD.

## 1 INTRODUCTION

Euclid's algorithm to compute the greatest common divisor (GCD) of two integers is one of the oldest algorithms known [7, sect. 4.5.2]. His algorithm describes a process that is inherently sequential, as are most algorithms typically used to compute the GCD, including those currently used by the the GNU Multiprecision Arithmetic Library (GMP) [6]. The modular integer GCD algorithm [12] is unique in that it employs a modular representation for the integer inputs and intermediate results in order to provide a way to parallelize the task. This paper describes an implementation [1] of the modular algorithm on a single NVIDIA graphics processing unit (GPU) [5]. Although its performance is inferior to the GCD operation provided by GMP, there is hope that it can be used as a building block for a multinode implementation that would provide superior performance on very large input values.

In the following section we first review the original modular algorithm, then present the modified version of the algorithm upon which the implementation is actually based. The following section discusses various aspects of the arithmetic operations needed to implement the algorithm. A description of thread synchronization needed by the implementation follows, after which comes a synopsis of the implementation's performance. The paper closes with plans for future work and a concluding summary.

## 2 ALGORITHM OVERVIEW

A corrected version of the basic algorithm, as given in [12], is provided in Figure 1. A simple typographic error in the original has been corrected on line 6. Lines 12 through 20 deal with a more significant error in the originally published version, which claimed that the final result can be computed as each mixed radix digit and modulus are obtained from the modular representation. In actuality, all the mixed radix digits and moduli must be computed before the recovery of the standard representation can begin, as is shown in the corrected version given here.

In the algorithm in Figure 1, $\pi(x, y)$ represents the number of primes in the interval $(x, y)$. A *symmetrical* modular representation is used to represent the integers as the algorithm is in operation, while the inputs are originally in a standard, positional, representation, so conversion to and from modular representation must be done. In symmetrical modular representation, values range between $-\lfloor q/2 \rfloor$ and $\lfloor q/2 \rfloor$, inclusive, for each modulus $q$, and all moduli are odd primes. Modular division is simply multiplication by the modular inverse: $u/v \bmod q = u \cdot v^{-1} \bmod q$, where $v^{-1}$

Authors' addresses: KENNETH WEBER, University of Mount Union, Department of Computer Science, 1972 Clark Ave. Alliance, OH, 44601, USA, weberk@mountunion.edu; JUSTIN A. BREW, University of Mount Union, Alumnus, Department of Computer Science, 1972 Clark Ave. Alliance, OH, 44601, USA, jbrew5662@gmail.com.

```
Input: Positive integers $U$ and $V$, with $U \geq V$
Output: $\gcd(U, V)$
$n \leftarrow \lfloor \log_2 U \rfloor + 1$
$w \leftarrow$ integer satisfying $\pi(2^{w-1}, 2^w) \geq \max\{\lceil 2^{w/2} + n \rceil, 9\}$
$Q \leftarrow$ the set of $\lceil 2^{w/2} + n \rceil$ largest primes $< 2^w$
forall $q \in Q$ do $[u_q, v_q] \leftarrow [U \bmod q, V \bmod q]$
repeat // Reduction loop
    $\mathcal{P} \leftarrow \{q \in Q \mid v_q \neq 0\}$
    $p \leftarrow$ element of $\mathcal{P}$ for which $|u_p/v_p \bmod p|$ is minimal
    $b \leftarrow u_p/v_p \bmod p$
    $Q \leftarrow Q - \{p\}$
    forall $q \in Q$ do $[u_q, v_q] \leftarrow [v_q, (u_q - b\, v_q)/p \bmod q]$
until $\forall q \in Q, v_q = 0$
$k \leftarrow 0, \;\; G \leftarrow 0$
repeat // Recover mixed-radix representation
    $k \leftarrow k + 1$
    $p_k \leftarrow$ an element of $Q$
    $g_k \leftarrow u_{p_k}$
    $Q \leftarrow Q - \{p_k\}$
    forall $q \in Q$ do $u_q \leftarrow (u_q - g_k)/p_k \bmod q$
until $\forall q \in Q, u_q = 0$
for $i = k$ downto $1$ do $G \leftarrow g_i + p_i G$
return $|G|$ // Return standard representation
```

Fig. 1. Corrected modular integer GCD algorithm

satisfies the equation $v \cdot v^{-1} \equiv 1 \mod q$, and can be computed with an extended greatest common divisor algorithm; see the discussion following Equation 23 from [7, p. 290] for more information.

The algorithm outlined in Figure 1 requires a very large number of moduli to guarantee that a final result can be computed. In most cases, it is possible to use significantly fewer moduli; the implementation we are describing uses the variant of the original algorithm given in Figure 2 to do just that. It estimates the number of moduli that will be needed, based only on the sizes of the inputs, and checks to see whether there will be enough moduli to finish the computation at each iteration of the reduction loop.

The formula on line 2 that computes the number of moduli needed $N_Q$, including the constant $C_L$, is heuristically determined; it is derived from the average number of bits each iteration of the reduction loop reduces the sizes of the intermediate values. In the actual implementation, $N_Q$ is made larger, if necessary, so that the number of moduli is a multiple of the thread block size.

Using this method to determine the number of moduli to use introduces a possible error in that there may not be enough moduli to recover the actual GCD, or even to exit the reduction loop. Such an error is detected by keeping track of the number of moduli currently active and comparing it to the number of moduli needed to recover the current values of the integers represented by $u_q$ and $v_q$ at each iteration of the reduction loop. The implementation fails if this error occurs.

```
**Input:** Positive integers $U$ and $V$, with $U \geq V$
**Output:** $\gcd(U, V)$
**Constants:** $L$ = integer $\geq 2$
               $\mathcal{M}$ = set of primes in the range $(2^{L-1}, 2^L)$
               $C_L = 1.6 - 0.015 \cdot L$
$N_u \leftarrow \lfloor \log_2 U \rfloor + 1,\ N_v \leftarrow \lfloor \log_2 V \rfloor + 1$
$N_Q \leftarrow \lceil C_L \cdot N_u / \log_{10} N_u \rceil$
if $N_Q > ||\mathcal{M}||$ then return fail
$Q \leftarrow$ the set of $N_Q$ largest elements of $\mathcal{M}$
forall $q \in Q$ do
    $[u_q, v_q] \leftarrow [U \bmod q, V \bmod q]$
    $t_q \leftarrow$ if $v_q = 0$ then $\infty$ else $u_q / v_q \bmod q$
end
$[p, b] \leftarrow$ [element $q$ of $Q$ for which $|t_q|$ is minimal, $t_q$]
repeat // Reduction loop
    $Q \leftarrow Q - \{p\}$
    forall $q \in Q$ do
        $[u_q, v_q] \leftarrow [v_q, (u_q - b \cdot v_q)/p \bmod q]$
        $t_q \leftarrow$ if $v_q = 0$ then $\infty$ else $u_q / v_q \bmod q$
    end
    $N_Q \leftarrow N_Q - 1,\ [N_u, N_v] \leftarrow [N_v, N_v - L + \lfloor \log_2 b \rfloor]$
    if $N_Q (L - 2) \leq N_u$ then // Can't recover $G$
        return fail
    $[p, b] \leftarrow$ [element $q$ of $Q$ for which $|t_q|$ is minimal, $t_q$]
until $b = \infty$
$k \leftarrow 1,\ G \leftarrow 0$
$[p_1, g_1] \leftarrow$ [element $q$ of $Q$ with priority to $u_q \neq 0, u_q$]
repeat // Recover mixed-radix representation
    $Q \leftarrow Q - \{p_k\}$
    forall $q \in Q$ do $u_q \leftarrow (u_q - g_k)/p_k \bmod q$
    $k \leftarrow k + 1$
    $[p_k, g_k] \leftarrow$ [element $q$ of $Q$ with priority to $u_q \neq 0, u_q$]
until $g_k = 0$
for $i = k - 1$ downto $1$ do $G \leftarrow g_i + p_i G$
return $|G|$ // Return standard representation
```

Fig. 2. Modular algorithm, as implemented

## 3 ARITHMETIC

All arbitrary-precision arithmetic on the CPU is done with GMP [6], including random generation of pairs of test values, which are exported from GMP as arrays of 32-bit integers and sent to the GPU. What follows is a discussion of the implementation of the basic arithmetic operations that must be performed on the GPU in order to implement the modular algorithm. Most of these operations are done on values in the range $[0, q - 1]$; it may be necessary to bring the result back into the symmetric range by subtracting $q$.

### 3.1 Computing remainders

A technique provided by Cavagnino and Werbrouck in [2] allows us to compute $x \bmod q$, where $x$ is a 64-bit value—usually a product of two 32-bit numbers—and $q$ is an $L$-bit modulus ($L \leq 32$), as one 64-bit × 64-bit ⇒ high 64 result bits multiplication, one 64-bit right shift, one 64-bit × 64-bit ⇒ low 64 result bits multiplication, and one 64-bit subtraction, rather than by performing 64-bit integer long division. This is significantly faster than the code generated by the C++ compiler for the 64-bit remainder operator `%`. A run-time constant for each modulus, essentially the "inverse" of the modulus, is calculated when the GPU kernel first starts executing.

Cavagnino and Werbrouck list four categories into which $x$ and $q$ could fall, two of which do not apply here because the divisor $q$ is always odd. For this implementation we only use moduli which fall into the category that requires the fewest operations to compute the remainder, thus forcing us to rule out a significant portion of $L$-bit primes. For instance, roughly 30 million of the 98 million 32-bit primes are not useable as moduli. By requiring somewhat more work at runtime per modulus [2, sect. 2.4], all $L$-bit primes could be used.

Finally, note that the arbitrary-precision remainder required by line 7 of Figure 2 is performed by repeatedly using the $x \bmod q$ operation discussed above.

### 3.2 Modular inverse

As noted earlier, the modular inverse can be computed with an extended greatest common divisor algorithm. A variation of the extended Euclidean algorithm [7, Algorithm 4.5.2X] is used in our implementation. Integer division is too slow to use to compute the quotient and remainder, so floating point reciprocal of the divisor is used to obtain a quotient that may sometimes may be one too small. The algorithm given in Knuth must be adjusted only slightly to accomodate this anomaly. Care must also be taken to ensure that warp divergence [5, Section 5.4.2] doesn't occur, which can happen when the loop control structures become too complicated. Details are provided in [11].

### 3.3 Finding moduli

The sieve of Eratosthenes [7, sect. 4.5.4, problem 8 and solution] is used as the foundation for a separate program that is run before compilation of the main program occurs. It discovers $L$-bit primes that satisfy the criteria needed to make the mod operation fast, as discussed in section 3.1, and produces a declaration for a static array that holds the moduli in CPU memory. These declarations are then incorporated into the source code of the main program as it is compiled. Once the main program determines the number of moduli needed, that many moduli are copied from CPU memory to GPU global memory.

## 4 PARALLEL CONSTRUCTS

In this section we describe the implementation of the basic parallel constructs needed to implement the modular algorithm.

### 4.1 Grid synchronization

It was originally necessary to implement our own global barrier to synchronize all threads executing in a grid; this custom barrier is described below. In CUDA 9, NVIDIA introduced Cooperative Groups to its programming model [5, Appendix C], providing a reliable means to synchronize all threads executing in a grid. Cooperative Groups are not available on some NVIDIA GPU architectures, and have proven to be somewhat slower than our custom synchronization on GPUs with the Pascal or Volta architectures. The implementation [1] compiles into two versions, one using our custom barrier, and one using Cooperative Groups. The times reported in Table 2 are from the version using our custom barrier.

The custom barrier relies on a $4 \times$ grid size array of 64-bit unsigned integers, allocated in device global memory, to communicate between all the streaming multiprocessors (SMs) in the device. At any given synchronization point, all the values in one row of the array are initially zero; each thread block will post a nonzero value to the slot indexed by its block identifier, then the thread block will assign one thread per slot in the row to perform a busy wait until the value in that slot is nonzero, at which time the thread stores the data from its slot in the array, resets the slot's value to zero, increments (circularly) the row index to the one it should look at next, then performs a synchronization with all other threads in the block. The nonzero value is treated as a (modulus, value) pair by other parts of the program, and handled somewhat differently depending on whether a global minimum or a global any is being computed; see sections 4.2 and 4.3, respectively.

In order to avoid deadlock, no more thread blocks than could be active at one time over all SMs are allocated; this is achieved with the help of the `cudaOccupancyMaxActiveBlocksPerMultiprocessor` library routine, which calculates the number of thread blocks each SM can handle, given a kernel and the number of threads per block needed to execute that kernel [9].

## 4.2 Global minimum

The global minimum needed on lines 9 and 19 of Figure 2 is computed as follows. First, each thread constructs a 64-bit unsigned value by concatenating $|t_q|$ with $q$, both 32-bit unsigned integers. A global minimum on 64-bit values is performed as a typical reduction operation in each thread block, using the warp shuffle mechanism provided in CUDA [5, Appendix B.15] together with streaming multiprocessor shared memory to disseminate the result between warps, to get a minimum for the thread block. Then each thread block posts its own minimum to the global barrier (described in the previous section) and waits for the results from all the other thread blocks. Once these are obtained, each thread block computes the minimum for itself; all thread blocks are guaranteed to get the same minimum pair, because the moduli are unique and included in the data being minimized.

## 4.3 Global any

The global "choose any" needed on lines 22 and 27 of Figure 2 is computed in a manner similar to the global minimum. described in the previous section, except that *any* pair a thread block has is posted to the global barrier, with preference given to a pair with a nonzero value. (Note that even a pair with a zero value will be nonzero because it contains a nonzero modulus, so that threads waiting on the barrier will stop waiting.) When each block chooses among all the values posted to the barrier, it is imperative that they all choose the same value; this is done using the`__ballot_sync` function [5, Appendix B.13].

## 5 RESULTS

In Figure 3 we graph the times required to execute test runs of the implementation on several NVIDIA GPUs for input pairs of several sizes. The clock frequency of each of the GPUs is listed in Table 1. We also graph the times required to perform the same operations using the GMP implementation of the GCD operation on a reference CPU, an Intel Xeon E5-2620 operating at 2.4 GHz. Table 2 provides the actual data upon which Figure 3 is based.

For all times reported, the executables were compiled with the value of $L = 32$. Ten pairs of inputs were randomly generated for each size reported, and saved to be used for tests on each device. The average execution time for the ten pairs is reported. A one-time initialization phase of about 300 ms (only 80 ms for the V100) is not included in the times reported in the table and graphs. Note that, since the pairs are randomly generated, the likelihood of any pair having a GCD larger than a few bits long is very low. Times reported are actual physical times needed to compute the values; thus some system activity may have affected the results slightly, although care was taken to reduce, as much as possible, any such activity.

Figure 4 highlights the behavior of the implementation on the Tesla V100. Given enough processing elements, the modular GCD algorithm should exhibit linear behavior [12]. It can be seen that the execution times exhibit linear behavior up to the point at which the device becomes saturated by the number of threads it must support, which appears at input sizes of 160 Kibit. Using the method of least squares over the range 16 to 160 Kibit on the timing data to estimate the constants gives us the equation $T = 0.35\,N + 1.62$, where $N$ is measured in Kibits. We use this extrapolation

Table 1. Clock frequency of GPUs (in GHz)

| Name | Frequency |
|---|---|
| GeForce GTX 980 Ti | 1.08 |
| GeForce GTX 1080 | 1.73 |
| Tesla P100 | 1.33 |
| Tesla V100 | 1.53 |

Table 2. ModGCD and GMP GCD execution times

| Input | GPU times (ms) | | | | CPU |
|---|---|---|---|---|---|
| size | GTX | GTX | Tesla | Tesla | time |
| (Kibit) | 980 Ti | 1080 | P100 | V100 | (ms) |
| 16 | 9 | 5 | 8 | 7 | 0.6 |
| 32 | 16 | 10 | 15 | 13 | 1.7 |
| 48 | 22 | 15 | 20 | 18 | 2.6 |
| 64 | 33 | 22 | 27 | 25 | 3.8 |
| 80 | 40 | 27 | 33 | 31 | 5.0 |
| 96 | 47 | 32 | 40 | 35 | 6.5 |
| 112 | 53 | 38 | 47 | 40 | 7.9 |
| 128 | 65 | 45 | 54 | 46 | 8.7 |
| 144 | 78 | 50 | 61 | 52 | 10.9 |
| 160 | 88 | 56 | 85 | 58 | 12.9 |
| 176 | 98 | 71 | 98 | 66 | 14.9 |
| 192 | 124 | 80 | 112 | 73 | 16.3 |
| 208 | 136 | 87 | 124 | 81 | 17.8 |
| 224 | 152 | 103 | 138 | 94 | 17.9 |
| 240 | 170 | 116 | 152 | 106 | 20.1 |
| 256 | 199 | 125 | 164 | 118 | 22.5 |
| 272 | 220 | 136 | 177 | 132 | 24.1 |
| 288 | 238 | 158 | 189 | 143 | 25.6 |
| 304 | 257 | 174 | 199 | 154 | 26.5 |
| 320 | 294 | 186 | 230 | 165 | 27.3 |
| 336 | 321 | 198 | 261 | 174 | 30.6 |
| 352 | 342 | 221 | 276 | 184 | 32.9 |

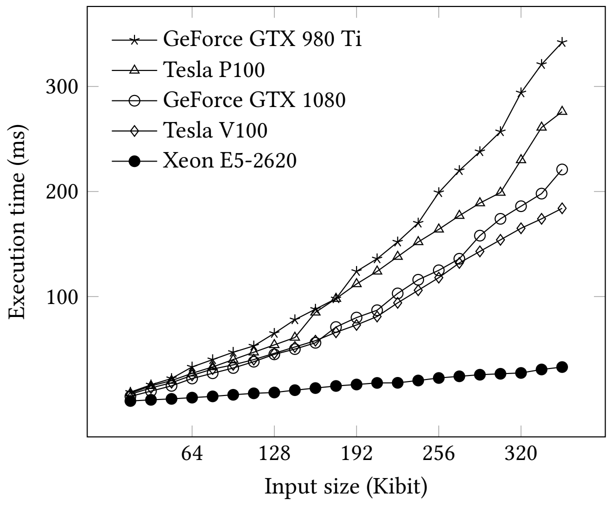

Fig. 3. Modular GCD times vs GMP GCD times

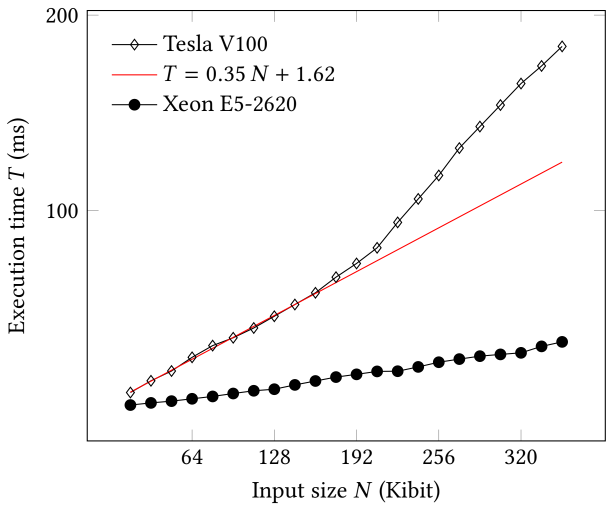

Fig. 4. Modular GCD behavior on Tesla V100

to estimate the amount of time needed by a multinode implementation of the algorithm, given enough GPU devices so that none of the devices are saturated. Although this is optimistic, in that it assumes that communication time between the processors will scale linearly, at least it is fairly realistic in terms of the arithmetic needed by the algorithm.

The GCD algorithms used by GMP for large input are essentially $O(N^{1+\epsilon}\log N)$ for a fairly large value of $\epsilon$ [6, section 15.3.3], so—given enough processing elements—a multinode implementation of the modular GCD algorithm should be faster than the GMP implementation for very large input. Comparing projected execution times for the modular algorithm against actual times recorded for the GCD operation from GMP, we predict that the modular algorithm will be faster than the GMP implementation for inputs of over 50 million bits, at which value GMP GCD took an average of 17 seconds for ten pairs of 50 million bit inputs, and our linear extrapolation projects a running time of around 17 seconds on a hypothetical multinode system with enough GPUs of the same type as the projection is based on, and with fast enough communication between GPUs.

A rough estimate of the number of GPUs needed to support the modular GCD for inputs in the range in which it would be faster than GMP can be determined by dividing the number of moduli needed by the number of moduli one GPU could support. (Again, this estimate optimistically assumes trivial communication between GPUs.) The number of moduli needed to compute the GCD of two 50 million bit numbers would be around 7.27 million, using the formula for $N_Q$ given in 2. From Figure 4, we estimate that each V100 GPU is capable of supporting at least 35,000 moduli without going into the superlinear range of execution times. Thus the number of GPUs needed to compute the GCD of input sizes of 50 million bits would be around 210.

The size of the inputs at which crossover occurs presents a question: are there enough 32-bit primes to support the implementation at input sizes of hundreds of millions of bits? In section 3.3 above, we state that the number of usable 32-bit moduli is around 68 million, which should handle input sizes of up to 529 million bits, based on the formula for $N_Q$ from Figure 2. This should be sufficient to demonstrate the efficacy of the modular algorithm. Relatively minor changes would allow use of all 98 million 32-bit primes as moduli, although with slower execution times, and could handle input sizes of up to 777 million bits. It may also be possible to use 33-bit moduli with special techniques that employ predominantly single precision arithmetic, but in order to make the implementation *truly* useful for extremely large input, it will be necessary to use larger moduli, which will require either computationally expensive double precision integer arithmetic, or an advance in GPU design that provides native 64-bit integer arithmetic operations.

## 6 FUTURE WORK

We intend to extend this software to run on multinode clusters of CPU/GPU pairs on the Owens supercomputer at the Ohio State University [4]. The Message Passing Interface [8] will be used for inter-device communication. This system does not have fast enough GPUs, nor enough of them, to actually demonstrate the efficacy of the algorithm, but it will allow for a basic implementation to be developed and tested.

In order to demonstrate efficacy, it will be necessary to execute the software on a system such as the Summit supercomputer at the Oak Ridge National Laboratory [10]. It has approximately 4,600 nodes, each with six NVIDIA Tesla V100 GPUs, which is orders of magnitude more than the 210 estimated in the previous section to be needed to compute the GCD for inputs of over 50 million bits.

## 7 CONCLUSION

The implementation of the modular integer GCD algorithm for one GPU, described above, allows us to provide a realistic projection of the algorithm's performance on a multinode system, and leads to the conclusion that a multinode implementation on a supercomputer-class system could outperform implementations of sequential algorithms, such as GMP [6], for large enough input. Although the projection encourages further investigation, only an actual multinode implementation of the modular algorithm will allow a definitive assessment of its efficacy.

## ACKNOWLEDGMENTS

The authors thank the Ohio Supercomputer Center [3] for access to a Tesla P100 GPU on the Owens system, as well as Anthony Rizzo for his contributions to initial software development, NVIDIA Corporation for the donation of a Tesla

C2070 GPU, and Seneca Data Distributors (a subsidiary of Arrow Electronics) for the donation of computer time for testing of early versions of the project.

## REFERENCES

[1] Justin Brew and Kenneth Weber. 2018. ModGCD-OneGPU. Retrieved September, 2018 from https://github.com/mountunion/ModGCD-OneGPU/tree/v1.3-alpha

[2] D. Cavagnino and A. E. Werbrouck. 2008. Efficient Algorithms for Integer Division by Constants Using Multiplication. *Comput. J.* 51, 4 (2008), 470–480.

[3] Ohio Supercomputer Center. 1987. Ohio Supercomputer Center. http://osc.edu/ark:/19495/f5s1ph73.

[4] Ohio Supercomputer Center. 2016. Owens supercomputer. (2016). http://osc.edu/ark:/19495/hpc6h5b1

[5] CUDA C Programming Guide 2018. CUDA C Programming Guide. Retrieved April 4, 2018 from http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf version 9.1.85.

[6] Torbjörn Granlund. 2016. *GNU MP: The GNU Multiple Precision Arithmetic Library* (6.1.2 ed.). Free Software Foundation.

[7] Donald E. Knuth. 1997. *The Art of Computer Programming* (3rd ed.). Vol. 2: Seminumerical Algorithms. Addison-Wesley, Reading, Massachusetts.

[8] Message Passing Interface 2009. MPI: A Message-Passing Interface Standard. Retrieved March 6, 2018 from http://mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[9] NVIDIA CUDA Runtime API 2017. NVIDIA CUDA Runtime API. Retrieved April 5, 2018 from https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf

[10] Summit ORNL 2018. System User Guide: Summit. Retrieved March 6, 2018 from https://www.olcf.ornl.gov/for-users/system-user-guides/summit/

[11] Kenneth Weber. 2018. Modular Inverse for GPU-based Modular Arithmetic Systems. (2018). In preparation.

[12] Kenneth Weber, Vilmar Trevisan, and Luiz Felipe Martins. 2005. A Modular Integer GCD Algorithm. *Journal of Algorithms* 54, 2 (February 2005), 152–167.